

## QLA: The Quick Linear Algebra Library

The Quick Linear Algebra (QLA) library is a collection of very fast and easy-to-use linear algebra algorithms that enable high-speed applications at all scales, from small to massive.

*What is it for?* Fast computation of the SVD, linear systems, etc., for most real-world scenarios. As a bonus, QLA is much more robust to noise and ill-conditioning than standard linear algebra routines.

*What is it **not** for?* Identity matrices, some matrices with i.i.d. random elements, and other matrices whose rows/columns are nearly mutually orthogonal and of roughly the same norm (these tend to be artificial constructs). Also, applications where machine precision is absolutely required. Little or no speedup occurs in these scenarios – but then, who wants to decompose the identity matrix?

QLA includes fast algorithms for:

- Singular value decomposition (SVD)
- Solving linear systems
- Least squares, linear regression (multiple, multivariate)
- Matrix inverse and pseudoinverse
- Principal component analysis (PCA)

These algorithms are based on cutting-edge work in numerical linear algebra (see reference section). They have shown consistent high performance across a wide variety of matrices and datasets arising in real-world usage. Speedups over standard linear algebra software (MATLAB/LAPACK) have ranged as high as 10,000x. In the past, acceleration of this magnitude was possible only with supercomputing hardware. QLA brings high performance to individual computers with no special hardware – the power of a supercomputer, contained in an algorithm. We think this opens up worlds of possibilities, and we're excited to partner with our users in exploring the potential of these new tools.

The organization of this document is as follows: we first quickly cover installation and basic usage in MATLAB, which should be enough to get started. We then give performance results illustrating the speed of QLA on a variety of real-world datasets, followed by deeper discussion of common questions, detailed specifications of the algorithms, and a list of references for further reading.

## **Contents**

Installation.....	3
Support.....	3
Quick Usage in MATLAB.....	4
Performance.....	6
Questions and Answers.....	8
Conclusion.....	11
Detailed Specifications.....	12
References.....	16

## Installation

Go to <http://massiveanalytics.com/download.html> and follow the instructions for download. The file you download will contain a single directory called 'qla' that includes all QLA files. Extract the 'qla' folder to a location of your choice, such as the MATLAB toolbox directory. Then add the 'qla' path to MATLAB's path list as follows:

```
>> addpath('<path to QLA>') % e.g., '<MATLAB path>/toolbox/qla'
```

If you purchased a license, run `activate_qla` and enter the license activation key at the prompt:

```
>> activate_qla
Enter QLA activation key:
```

If you don't yet have a license, QLA will automatically enter a fully functional 30-day trial period. Your license activation key(s) can be entered at a later time using `activate_qla` as above.

Verify correct installation by running `verify_qla`, which should print a success message:

```
>> verify_qla
QLA successfully installed!
```

## Support

For questions or help, please contact [support@massiveanalytics.com](mailto:support@massiveanalytics.com). We pride ourselves on providing fast and personalized support.

## Quick Usage in MATLAB

Using QLA in MATLAB is simple: just add a `q` in front of your favorite linear algebra routine. For example, to compute the quick SVD of a matrix `A`, just type:

```
[U, S, V] = qsvd(A)
```

That's it, just add a `q` to the function name. The idea is for QLA to be an easy drop-in replacement for the standard linear algebra routines, but with orders of magnitude greater speed. The SVD is a good starting point because it can be used to compute so many other quantities of interest. The function `qsvd` can be used directly in such computations. (Heads-up: testing on random or identity matrices will give little speedup – try matrices that reflect more realistic usage instead.)

Here are other examples of using functions from QLA; each can be used as a drop-in replacement for the corresponding MATLAB function:

```
% solving a linear system: find X satisfying AX = B
```

```
X = qlinsolve(A,B)
```

```
% least-squares solution to AX = B
```

```
X = qlsqr(A,B)
```

```
% pseudoinverse of rectangular A
```

```
A_inv = qpinv(A)
```

```
% linear regression: fit B to model XB=Y
```

```
B = qlinreg(X,Y)
```

```
% PCA: project centered rows onto principal components
```

```
X_pca = qpca(X)
```

Now, before you try `qsvd` on `rand(1000,1000)`, remember that QLA is not expected to accelerate certain artificial cases such as uniformly random or identity matrices. Big speedups are observed with the kinds of matrices that arise in real-world usage – sensor data, images, simulations, control systems, scientific datasets, survey data, etc. QLA will be correct in all cases, and fast in most of them.

The "quick" in QLA means our algorithms are both fast and mildly approximate. Approximation quality is tightly controlled by conservative default settings so that QLA can be used as a drop-in replacement for traditional linear algebra routines. Additionally, when required, the `epsilon` error tolerance (a single number between 0 and 1, representing the maximum allowed relative squared error) can easily be set to as high or low a level as desired to obtain the optimal speed/quality tradeoff. Our final example shows how to override the default error tolerance to gain greater speed. For all QLA functions, this is accomplished by specifying a relative squared-error tolerance as the final parameter:

```
[U,S,V] = qsvd(A, 0.05) % override default 0.01 error tolerance
```

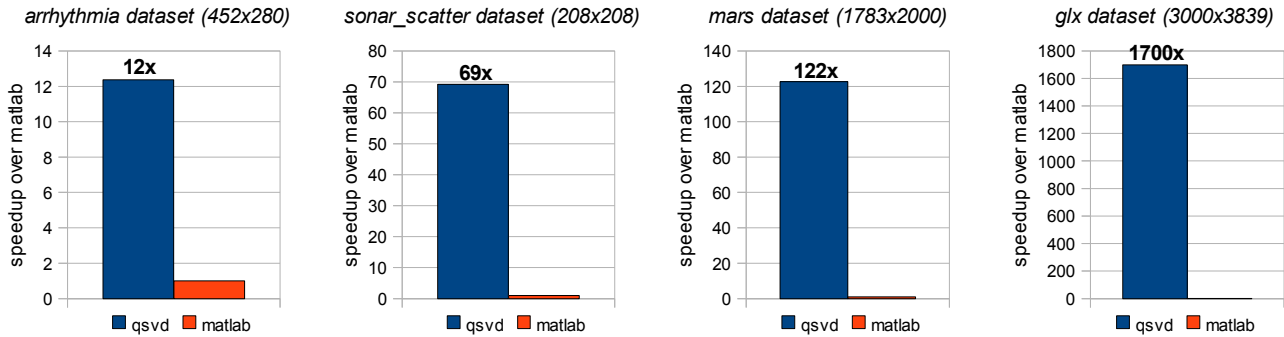
We provide full descriptions of all QLA functions in the Detailed Specifications section. Additionally, the QLA installation includes several demo scripts that showcase usage and performance on real datasets. Running the demos will show both performance statistics and, in the case of `qpca`, graphical comparison to the equivalent MATLAB routines. Feel free to take a look at the script code as well – you'll see that our usage of QLA is just as simple as in the examples we've shown. The demos can be found in the QLA installation directory, and can be run with the following commands:

- `demo_qsvd`: SVD on several datasets, focusing on speed comparison to MATLAB.
- `demo_qlsqr`: least squares solution of several systems, comparing speed and robustness to MATLAB. Representative of `qlinsolve` and `qlinreg` performance as well.
- `demo_qpca`: PCA on several datasets, with graphical and speed comparisons to MATLAB.

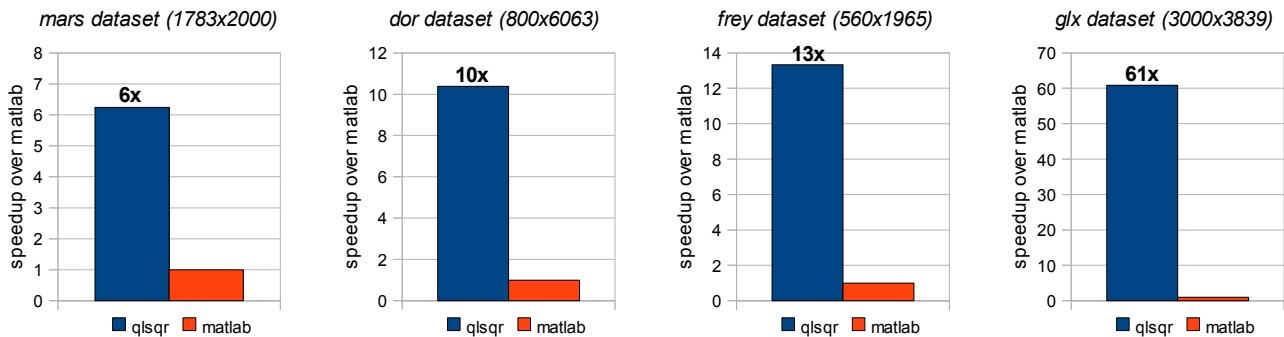
Again, remember that these methods provide significant acceleration in most scenarios, but there are a few exceptions, such as the identity matrix and certain i.i.d. random matrices. So if you run on `rand(1000,1000)`, don't be surprised if the acceleration isn't large – it's not expected to be. QLA is designed for the more common types of matrices encountered in real-world usage.

## Performance

Here we illustrate QLA performance with numerical comparisons to MATLAB on a variety of real data matrices with dimensions ranging from the low hundreds to the low thousands. All experiments were run at the default QLA error tolerances. Note that most of these comparisons are replicated in our demos, so both the data and the methodology can easily be examined.



**Fig. 1. `qsvd`: average speedup over MATLAB for matrices of various sizes.**



**Fig. 2. `qlsqr`: average speedup over MATLAB for matrices of various sizes.**

The first set of results compare QLA's `qsvd` to MATLAB's `svd` (which in turn is a LAPACK wrapper). We ran MATLAB's `svd` with the `'econ'` argument to ensure fair comparison. Performance is summarized in Fig. 1. Speedups for `qsvd` range from 12x to as high as 1700x.

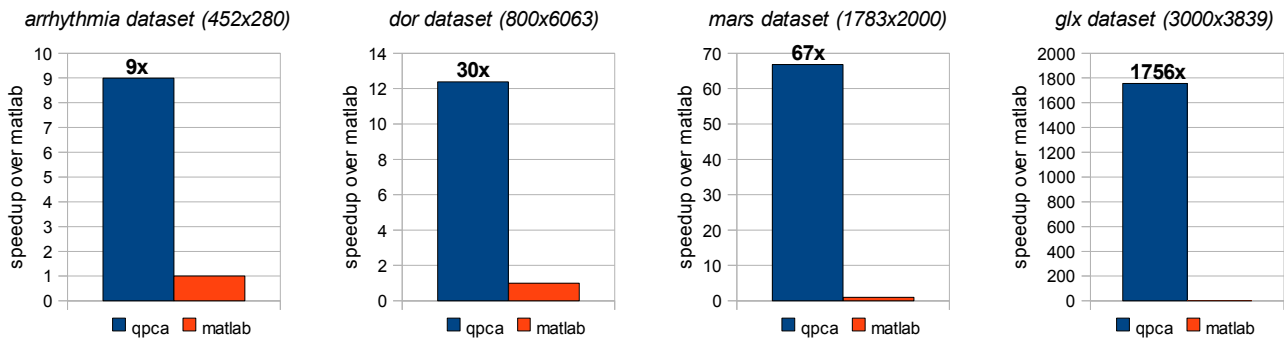
Next, we measure speed and robustness for QLA's `qlsqr` against MATLAB's built-in solver (the `'\'` operator). Robustness is measured by the RMSE error of  $(B-AX)$  on a held-out set of rows. These results are also representative of `qlinsolve` and `qlinreg`. Fig. 2 shows speedups for `qlsqr` ranging between 6x and 61x. Table 1 summarizes hold-out error for `qlsqr` vs. MATLAB. In each case

`qlsqr` has lower error by a factor of at least 2, and in one case by a factor of 23. This indicates that `qlsqr` can find substantially more robust solutions than MATLAB.

dataset	qlsqr rmse	MATLAB rmse
<i>mars</i>	14.29	57.21
<i>dor</i>	0.59	13.6
<i>frey</i>	17.67	35.11
<i>glx</i>	3.25	13.42

**Table 1. Average hold-out error (rmse) for qlsqr and MATLAB solver.**

Lastly, we compare the speed of QLA's `qpca` to an efficient MATLAB implementation. Fig. 3 shows the results: `qpca` speedups range from 9x up to 1756x over MATLAB.



**Fig. 3. qpca: average speedup over MATLAB for matrices of various sizes.**

An important trend to note is that speedup increases with the size of the input matrix. This holds in general, and agrees with theoretical results – all else being equal, the larger the matrix, the greater the speedup. Many applications deal with matrices much larger than those shown here, and will see correspondingly higher acceleration.

Though we generated these results at default error tolerances, greater acceleration would be observed if we set the tolerances higher, and less acceleration if we set them lower. Acceleration also depends to a degree on the amount of spectral concentration present in the matrix. See the next section for more detail on this point. Of course your mileage will vary depending on your data, but we've found these results to be fairly representative of performance on common varieties of data.

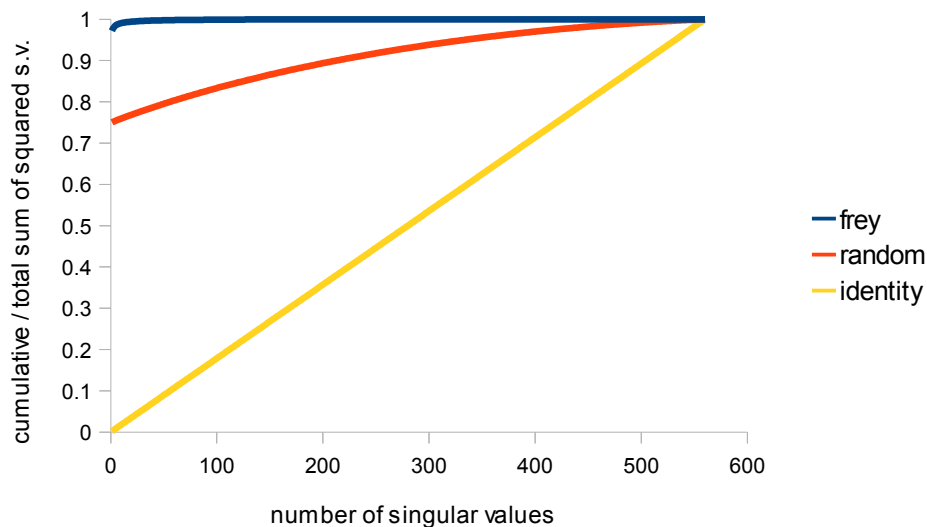
## Questions and Answers

This section covers some common questions. For uncommon questions, visit the support forum.

*What conditions are required for speedup?*

If i.i.d. random and identity matrices receive little or no speedup, how do I know which matrices will run fast? The short answer is that most will. The more technical answer is that matrices with more *spectral concentration* receive greater acceleration. In our experience, most real-world data have significant spectral concentration and therefore are significantly accelerated.

To illustrate, consider Fig. 4, which displays a measure of spectral concentration for three matrices: frey, a 560x1965 face recognition dataset, random, a 560x1965 matrix with uniformly random entries between 0 and 1, and identity, the 560x560 identity matrix. For each matrix, we order the singular values from largest to smallest and plot the partial sum  $\sum_{i=1}^k \sigma_i^2$  divided by the total sum  $\sum_{i=1}^{560} \sigma_i^2$  at each value  $k=1..560$ . This measures how much of the total singular value mass is concentrated in the first  $k$  singular values, i.e., the spectral concentration. Matrices with high spectral concentration have high-fidelity representations at lower rank than matrices with broader spectra, and our algorithms can exploit this to greatly accelerate computations.



**Fig. 4. Spectral concentration for real vs. artificial data.**

We can see from Fig. 4 that the frey dataset has high spectral concentration, as relatively few singular values are required to capture 99% or more of the total singular value mass. The random matrix, however, approaches the 99% mark much more slowly, while the identity matrix (with a maximally unconcentrated spectrum) reaches 99% last of all. This explains why the frey dataset runs very fast in QLA, while random is only slightly accelerated and identity not at all.

Our experience across a wide range of usage is that real-world matrices typically have enough spectral concentration to be accelerated in QLA. Matrices that receive little or no speedup are characterized by rows/columns that are all nearly mutually orthogonal and of similar norm – this is what makes their spectra broad and unconcentrated. Identity and permutation matrices are extreme examples; some very sparse matrices also share this property and are not well suited for QLA. Yet sparse matrices can turn out to have a surprising degree of exploitable structure, and diagonal matrices with significant differences among the diagonal entries can receive high acceleration. So it pays to try and see.

#### *Is it accurate enough?*

QLA algorithms perform tight approximations with strict, controllable error tolerance. Different applications will naturally tolerate different levels of approximation error. For ease of use we provide default error tolerances that we have found to work well across a diverse set of use cases. You may wish to override these in your own usage. Just keep in mind that speed is linked to error tolerance – the higher you can set `epsilon`, the faster the algorithms will run, but this should be balanced against the implications of looser approximation in your applications.

If you absolutely must fit down to machine precision, then QLA may not be what you need. But try and see – not only are QLA's approximations tight, they are also more robust to noise and ill-conditioning than traditional linear algebra algorithms, so you may find that QLA's results outperform those of traditional algorithms in your applications.

*Why are the outputs not full rank?*

You will notice that QLA algorithms produce results of lower rank than exact methods. This is expected, and comes from the reduced-rank representations used to accelerate computation. The key here is that the important matrix *dimensions* still match up, so there is no problem in using the results. For instance, `qsvd` on an  $m \times n$  matrix of rank  $n$  might return  $U, S, V$  that are  $m \times k, k \times k,$  and  $n \times k,$  respectively, with  $k < n$ . Though  $k$  is less than the full rank  $n$ , the dimensions still match up such that  $USV'$  is  $m \times n$ , as expected.

In addition to speedup, rank reduction provides a second benefit in many applications: it can filter out noise and stabilize computations on ill-conditioned matrices. So, for instance, you may find that solutions from `qlinsolve` and `qlsqr` work better than those of an exact solver, or that `qlinreg` avoids fitting process noise and has better out-of-sample accuracy than exact regression.

*How big must a matrix be for QLA to be helpful?*

Generally speaking, for significant speedup we expect the smallest matrix dimension to be at least 10, though we have observed speedup with even smaller dimensions. Also, among matrices with similar spectral concentration and number of entries, those that are squarer will produce higher acceleration.

One final note: an approximate inverse at low precision will of course not produce exact identity when multiplied by the input matrix – just a point to keep in mind when experimenting with `qinv`.

## Conclusion

We hope QLA makes a difference for you. We have found that its orders-of-magnitude acceleration enables not just faster performance, but entirely new usage – we can ask questions that were once too expensive to ask, perform analyses that were impractical to compute, and handle data at sizes and frequencies that were impossible before. Because QLA is an evolving product, we actively solicit your feedback to help us shape it to better suit your needs. Let us know what you think on our blog (<http://massiveanalytics.com/blog>), or drop us a line at [open.mic@massiveanalytics.com](mailto:open.mic@massiveanalytics.com). Good luck.

## Detailed Specifications

Here we provide full detail on the parameters and usage of all QLA functions. Note that square brackets denote optional arguments, e.g., `[epsilon]`. Sparse matrices must be converted to dense form.

### `qsvd(A, [epsilon=0.01])`

<b>Input</b>	<b>A</b> : a real matrix of any size <b>[epsilon]</b> : a relative error tolerance between 0 and 1; defaults to 0.01
<b>Output</b>	<b>[U, S, V]</b> : <b>U</b> : left singular vectors as orthonormal columns <b>S</b> : singular values on the diagonal <b>V</b> : right singular vectors as orthonormal columns <b>U, S, V</b> are guaranteed to satisfy $\ A - USV^T\ _F^2 \leq \epsilon \ A\ _F^2$
<b>Description</b>	Returns an SVD that reconstructs to within <code>epsilon</code> relative squared error of the input matrix. <code>U, S, V</code> satisfy the standard SVD properties. Runtime depends on <code>epsilon</code> and will be faster for larger values of <code>epsilon</code> .
<b>Examples</b>	<code>[U, S, V] = qsvd(A)</code> <code>[U, S, V] = qsvd(A, 0.1)</code>

### `qlinsolve(A, [epsilon=0.01])`

<b>Input</b>	<b>A</b> : a real $m \times n$ coefficient matrix <b>B</b> : a real $m \times p$ matrix of target values <b>[epsilon]</b> : a relative error tolerance between 0 and 1; defaults to 0.01
<b>Output</b>	<b>X</b> : a real $n \times p$ matrix satisfying $AX \approx B$ . Exact equality will hold for an approximate coefficient matrix <code>A'</code> satisfying $\ A - A'\ _F^2 \leq \epsilon \ A\ _F^2$ .
<b>Description</b>	Solves the linear system(s) $AX=B$ . Solution is exact for a high fidelity approximation <code>A'</code> of <code>A</code> . Because <code>A'</code> is formed by projecting <code>A</code> onto a subspace of its row space, the solution satisfies constraints very close to the original <code>A</code> and is also robust to noise in the coefficients. Runtime depends on <code>epsilon</code> and will be faster for larger values of <code>epsilon</code> .
<b>Examples</b>	<code>x = qlinsolve(A, b)</code> <code>% mean sq. error of solution</code> <code>mse = mean((A*x - b).^2)</code>

### qlsqr(A, [epsilon=0.01])

<b>Input</b>	<b>A</b> : a real $m \times n$ coefficient matrix <b>B</b> : a real $m \times p$ matrix of target values <b>[epsilon]</b> : a relative error tolerance between 0 and 1; defaults to 0.01
<b>Output</b>	<b>X</b> : a real $n \times p$ least-squares solution to $AX \approx B$ . Exact equality will hold for an approximate coefficient matrix $A'$ satisfying $\ A - A'\ _F^2 \leq \epsilon \ A\ _F^2$ .
<b>Description</b>	Approximate least-squares solution of the linear system(s) $AX=B$ . Solution is exact for a high fidelity approximation $A'$ of $A$ . Because $A'$ is formed by projecting $A$ onto a subspace of its row space, the solution satisfies constraints very close to the original $A$ and is also robust to noise in the coefficients. Runtime depends on <code>epsilon</code> and will be faster for larger values of <code>epsilon</code> .
<b>Examples</b>	<pre>x = qlsqr(A,b) % mean sq. error of solution mse = mean((A*x - b).^2)</pre>

### qlinreg(X, Y, [epsilon=0.01])

<b>Input</b>	<b>X</b> : a real $m \times n$ matrix with data points as rows <b>Y</b> : a real $m \times p$ matrix with scalar or vector targets as rows <b>[epsilon]</b> : a relative error tolerance between 0 and 1; defaults to 0.01
<b>Output</b>	<b>B</b> : a real $n \times p$ weight vector or matrix such that $XB \approx Y$ . Exact equality will hold for an approximate data matrix $X'$ satisfying $\ X - X'\ _F^2 \leq \epsilon \ X\ _F^2$ .
<b>Description</b>	Performs (multiple, multivariate) linear regression of the target $Y$ against the inputs (design matrix) $X$ . The regression is exact for a high fidelity approximation $X'$ of $X$ . This yields a regularizing effect that often filters out noise and produces significantly better out-of-sample performance. Runtime depends on <code>epsilon</code> and will be faster for larger values of <code>epsilon</code> .
<b>Examples</b>	<pre>B = qlinreg(X,Y) Y_pred = X_new * B % predict out-of-sample targets mse = mean((Y_new - Y_pred).^2) % mean sq. error</pre>

**qinv(A, [epsilon=0.01])**

<b>Input</b>	<b>A</b> : a real, square matrix of any size <b>[epsilon]</b> : a relative error tolerance between 0 and 1; defaults to 0.01
<b>Output</b>	<b>A<sup>-1</sup></b> : approximate inverse of <b>A</b>
<b>Description</b>	Returns an approximation to the inverse of a square matrix. The returned inverse is exact for a high fidelity approximation <b>A'</b> satisfying $\ A - A'\ _F^2 \leq \epsilon \ A\ _F^2$ . Note that in some use cases, greater efficiency may be obtained by keeping the inverse implicit in the form of an SVD so subsequent matrix/vector products can be ordered for maximum speed. For instance, if we need to compute $A^{-1}x$ , it may be faster to compute the SVD $A \approx USV^T$ followed by $V(S^{-1}(U^T x))$ than to explicitly compute $A^{-1}$ followed by $A^{-1}x$ .
<b>Examples</b>	<code>A_inv = qinv(A)</code> <code>A_inv = qinv(A, 0.005)</code>

**qpinv(A, [epsilon=0.01])**

<b>Input</b>	<b>A</b> : a real matrix of any size <b>[epsilon]</b> : a relative error tolerance between 0 and 1; defaults to 0.01
<b>Output</b>	<b>A<sup>+</sup></b> : approximate pseudoinverse of <b>A</b>
<b>Description</b>	Returns an approximation to the Moore-Penrose pseudoinverse. The returned pseudoinverse is exact for a high fidelity approximation <b>A'</b> satisfying $\ A - A'\ _F^2 \leq \epsilon \ A\ _F^2$ . Note that in some use cases, greater efficiency may be obtained by keeping the pseudoinverse implicit in the form of an SVD so subsequent matrix/vector products can be ordered for maximum speed. For instance, if we need to compute $A^+x$ , it may be faster to compute the SVD $A \approx USV^T$ followed by $V(S^{-1}(U^T x))$ than to explicitly multiply $VS^{-1}U^T \approx A^+$ followed by $(VS^{-1}U^T)x$ .
<b>Examples</b>	<code>A_pinv = qpinv(A)</code> <code>A_pinv = qpinv(A, 0.005)</code>

`qpca(A, [epsilon=0.05])`

<b>Input</b>	<p><b>A</b>: a real matrix of any size with data points as rows  <b>[epsilon]</b>: a relative error tolerance between 0 and 1; defaults to 0.05</p>
<b>Output</b>	<p><b>[X]</b>:  <b>x</b>: PCA-transformed data, i.e., centered and projected  <b>[V,D,mean]</b>:  <b>V</b>: principal components as orthonormal columns  <b>D</b>: diagonal matrix of principal component variances/eigenvalues  <b>mean</b>: mean of the input data as a row vector, needed for projecting new data points onto the principal components (e.g., <math>(x - \text{mean}) * V</math>)  <b>[X,V,D,mean]</b>: includes all return values described above  Principal components are guaranteed to capture <math>1 - \text{epsilon}</math> of the centered data variance. At the default <code>epsilon(0.05)</code>, at least 95% of variance is captured. Formally, <math>\ A_C - A_C V V^T\ _F^2 \leq \epsilon \ A_C\ _F^2</math>, where <math>A_C</math> represents <math>A</math> after centering.</p>
<b>Description</b>	<p>Returns a set of principal components and associated variances/eigenvalues, the mean of the data rows, and/or the PCA-transformed data. Runtime depends on <code>epsilon</code> and will be faster for larger values of <code>epsilon</code>.</p>
<b>Examples</b>	<pre>X = qpca(A) [V,D,mean] = qpca(A, 0.05) % transform new data points contained in B B_pca = (B - repmat(mean,size(B,1),1)) * V</pre>

## References

1. D. Achlioptas and F. McSherry. Fast Computation of Low Rank Matrix Approximations. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, 2001.
2. M. Brand. Fast Online SVD Revisions for Lightweight Recommender Systems. In *SIAM International Conference on Data Mining*, 2003.
3. A. Deshpande and S. Vempala. Adaptive Sampling and Fast Low-Rank Matrix Approximation. In *10th Int. Workshop on Randomization and Computation (RANDOM)*, 2006.
4. A. Deshpande, L. Rademacher, S. Vempala, and G. Wang. Matrix Approximation and Projective Clustering via Volume Sampling. *Theory of Computing*, 2:225–247, 2006.
5. P. Drineas, E. Drinea, and P. S. Huggins. An Experimental Evaluation of a Monte-Carlo Algorithm for Singular Value Decomposition. *Lecture Notes in Computer Science*, 2563:279–296, 2003.
6. P. Drineas, R. Kannan, and M. W. Mahoney. Fast Monte Carlo Algorithms for Matrices II: Computing a Low-Rank Approximation to a Matrix. *SIAM Journal on Computing*, 36(1):158–183, 2006.
7. S. Friedland, A. Niknejad, M. Kaveh, and H. Zare. Fast Monte-Carlo Low Rank Approximations for Matrices. In *Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering*, 2006.
8. A. Frieze, R. Kannan, and S. Vempala. Fast Monte-Carlo Algorithms for Finding Low-Rank Approximations. *Journal of the ACM (JACM)*, 51(6):1025–1041, 2004.
9. M. P. Holmes, A. G. Gray, and C. L. Isbell, Jr. QUIC-SVD: Fast SVD Using Cosine Trees. In *Advances in Neural Information Processing Systems (NIPS) 22*, 2009.
10. E. Liberty, F. Woolfe, P. Martinsson, V. Rokhlin, and M. Tygert. Randomized Algorithms for the Low-Rank Approximation of Matrices. *Proceedings of the National Academy of Sciences*, 104(51):20167–20172, 2007.
11. P. Martinsson, V. Rokhlin, and M. Tygert. A Randomized Algorithm for the Approximation of Matrices. Tech. report, Yale University Dept. of Comp. Sci., June 2006.
12. T. Sarlós. Improved Approximation Algorithms for Large Matrices via Random Projections. In *47th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 143–152, 2006.